18-644 Fall 2013 Project: GPS Bus Tracker

Edward Sears <esears@andrew.cmu.edu> Grant Skudlarek <gskudlar@andrew.cmu.edu>
Kory Stiger <kstiger@andrew.cmu.edu>

Abstract

In this paper, we will propose and discuss a novel way to track busses and accurately predict arrival times. We start by formally stating the problem and its importance. We then list the competition and their subsequent limitations. Next, we describe our idea and discuss our proof of concept implementation including important algorithms. Lastly, we describe the hardware needed to make the system fully functional.

1. The Problem

Public transporation can be an important asset to any city. If widely used, it can improve air quality, reduce traffic congestion, and more importantly provide transportation for low income families [4] and those who do not have cars. However, in Pittsburgh (and many other cities across the country) the public transportation is very unreliable. The bus system posts schedules, but can rarely stick to them. This can cause up to an hour delay between the scheduled time and the bus arrival time. For many people, this uncertainty makes the bus system practically unusable. Because of this, fewer and fewer people are using public transportation regularly. This results in the bus company earning less revenue and consequently reducing the number of bus routs. If this trend continues, bus companies will not be able to stay in business, leaving many people without any mode of transportation.

2. Importance

Making public transportation more user friendly could entice many more people to use it. This would not only increase revenues for the bus company, but would create a valuable resource for the residents of the city. More people could rely on the bus system on a regular basis and could increase the quality of living in the city in general.

3. Competition

Name	**Limitation**
Google Maps	Only posts scheduled times
CTA Bus Tracker	Chicago specific (not easily expandable)[2]
Tiramisu	Relies on individuals to report bus activity (crowdsourced) [3]
Maimi-Dade Bus Tracker	Miami Specific, Only tracks a single route

4. Our Idea

Our system aims to predict bus arrival times via GPS tracking of all buses in Pittsburgh. GPS modules placed on each bus would transmit the buses coordinates to a server application via cell networks. The server could then serve bus location data to users via the internet, who may be accessing the data through their mobile devices.

While GPS data will let users know where buses are, it may not let them know how long it will take for the bus to arrive at their stop. The user will not be able to know if the bus is stuck in traffic, or if it will arrive within a reasonable amount of time. The Pittsburgh bus schedule already shows how long it usually takes for a bus to travel between stops. In the absence of traffic, this data can be used to determine when the bus will arrive, based on its current position along the route.

In order to give users accurate results regardless of conditions, traffic patterns must be gathered in real time, or be based on a record of past traffic patterns. One approach which we are considering taking is the use of Google traffic data. By querying Google traffic data in real time along the buses route, the appropriate traffic delay can be added based on the situation. The other approach we are considering would be to accumulate a model of traffic patterns over a time period. This data, accumulated from past bus runs, would allow arrival time on a route to be predicted during a certain time of day. The actual position data and journey times of buses could be used to make the model more accurate. For the time period used for the model, longer time periods would allow more accurate predictions, but shorter time periods allow for faster adaption and deployment. We've determined that the best compromise would be week-long cycles. In our proof of concept design, we have chosen to use the Google Maps estimation technique.

The hardware module requires a GPS receiver, a processing unit, and a cellular modem. For our proof of concept, we will use an Arduino for the processing unit. For the modem component, we will use an inexpensive cell phone connected to a circuit board which can access the cellular modem via the phone serial port. The hardware setup is described explicitly and in detail in a later section.

5. Benefits of Our Idea

Our idea is more effective than current methods for 3 main reasons:

- Reliability: Because our system uses real hardware instead of relying on crowdsourcing techniques, the quality of service does not vary by time of day or by popularity.
- Easy Set Up: Our system can leverage the bus route data

that is curretly present in Google Maps. Because of this, our system can be easily ported to different cities and different public transportation modes.
- Accurate Prediction: Our system will have the ability to accurately predict arrival times for busses by leveraging current position data. Arrival times will be updated in real time in order to give up to date information of the arrival times of various buses passing by a user's location.

6. Implementation

6.1. Front-End

The options for the implementation of the front end interface were to make a native smartphone application or to make a web-based application. The web-based implementation was chosen over a native smartphone app for a couple reasons. Firstly, there were no real benefits to using a native application in this situation. Due to the nature of this application, it relies on accessing external resources such as the Google Maps API and communication with a central server. Thus it made sense to have the entire package built around using HTML and javascript which meant easy integration with the services mentioned above. The extra time it would have taken to make a native application would have been unnecessary and ultimately would have limited the user base to one type of smartphone. Using a web-based approach meant that the application would be reachable by the maximum number of users - even including anyone with a simple internet connection in addition to those using a smartphone.

6.1.1. Setup The actual setup of the UI (or front end application), was quite simple. The HTML portion of it was kept as simple as possible with just a few buttons and an area to hold the Google Map. The malfunctioning hardware which was supposed to send position data for each bus to the server was temporarily replaced by a button - this is the "Tracked Point" button. Thus, upon clicking this button the application will simulate the sending of a set of latitude/longitude coordinates to the server to be added to a list for that specific bus. The other important button is the one which says "Find Buses". After designating the position of interest on the map, the user will then push this button and a list of arriving buses along with their ETA will be displayed. Behind the scenes, there are also a few uses of the Google Maps API in order to visually display what is happening. Upon load of the application, a new map is created centered on Pittsburgh, then the user's current position is loaded as a green marker. When the user clicks (or taps) on the map, a red marker will appear and its position information is what is used in conjunction with the buttons below the map. When the user requests to "Find Buses", an API call is made to determine the estimated time of arrival of the buses.

6.1.2. Challanges The most interesting discovery after working on this part deals with how the Google API estimated time. The first attempt to get time estimations involved sending a set of waypoints in order to hopefully keep the route on the right path for the specific bus it was concerned about. However, in order to send waypoints, the call to the API needed to specify that it was using a route driven by a normal vehicle. This method returned a surprisingly inaccurate time even though each point along the route was specified. As it turns out, simply using a start and end point while specifying the travel option as public transit resulted in a much more accurate time estimation even though there were no middle points involved to ensure that the route was moving along the right path.

6.2. Back-End

After much deliberation, the options for the implementation of the back end were narrowed down to the best option being a central server hosting a database. There were obviously many options for the type of server and database, however the team decided to use Node.js and MongoDB because of previously gained expertise. In addition, this setup was beneficial because MongoDB easily plugs into a Node.js server and Node.js is written in javascript so it is very easy to work with for web applications.

6.2.1. Setup The two most important parts about the back end is the implementation of the database to hold the tracked bus points and the inclusion of the bus tracking algorithms. At a high level, the mongo database has two collections - one collection for the list of known bus routes and one collection for the list of buses being tracked. The known bus routes are stored according to their actual name (ie: 61C, 67, etc), while the tracked bus routes are stored according to a unique identifier for each bus. Also, in order to provide for the occurrence where the server inevitably crashes, all of the known bus routes have been pre-recorded and are thus automatically stored at the start of each reboot. For simplicity's sake, the few known bus routes with which testing was done were simply hard coded at the end of the server file in order to avoid having to debug the linking errors sure to arise from including multiple files.

6.2.2. Challanges As it turns out, every database access, whether it be an update, search, or save, is fully asynchronous. This made it difficult to preserve the flow of the program since the asynchronous calls would be made then the functions would return before having the proper variables set from within the asynchronous callback functions. In order to work around this, one would think that a simple while loop at the end of the function could 'wait' on a variable set in the callback function before allowing the function to return; however, javascript only uses one thread so this sort of coding would result in an infinite loop without every allowing the asynchronous call to run or return. The work around which fixed this problem involved nested asynchronous callbacks and a global counter to keep track of the number of expected callbacks before allowing the function to return.

7. Algorithms

Our tracking system is based on 3 fundamental algorithms. While only the first two listed below were used in the final implementation, all 3 were implemented. If future versions of the bus tracking system were implemented, all 3 algorithms would surely be exercised. Below describes each in detail.

In this section, a "tracked point" will be a pair of GPS coordinates that was sent from a bus tracker. A "route point" will be a GPS coordinate that is known to be along a specified route.

7.1. Tracked Bus to Bus Route

This algorithm dynamically maps each GPS tracked bus to a known bus route. This dynamic mapping reduces the setup for our GPS tracking device to merely placing the unit on the bus.

Each time a gps coordinate is sent from a bus tracker to the main database, this algorithm calculates the potential bus routes the bus could be following. The algorithm computes the potential matches using these steps:

1. Pick a known route
2. Find closest route point for each tracked point a bus has.
3. Add the difference of that point to a running total for each known route
4. repeat 1-3 for all known routes

This will output a minumum total difference between the tracked points we have and each known route. Intuitively, the known route that has the lowest total difference would be the estimated route for the bus. To account for small variations in the known route points, we included a threshold for the route mapping. This threshold allows the algoritm to pick multiple bus routes as potential matches if the bus' path is currently contained in multiple routes. Once there is a difinitive difference in the route a bus is following, the algorithm will output only one bus route name. To illustrate this further, we can imagine the extremes. When a bus transmits its first GPS coordinate right outside of the bus depot, the algoritm will likely output all bus routes as a potential match. At the completion of an entire route loop, the algorithm will choose a single route as the match in the worst case. The results section will describe this in more detail.

7.1.1. Results In evaluating the effectiveness of this algorithm, we utilized a large database of bus traces from Seattle [1]. After adding 5 overlapping routes, we added tracked points until the algorithm chose the correct known bus route prediction. In these trials, the algorithm chose the correct bus route in an average of about a .002 difference in lattitude and longitude coordinates from the point of divergence. This translates to the bus diverging for about 250 yards. Due to the threshold calculations, the algorithm takes slightly more divergent distance the fewer points that have been logged.

7.1.2. improvements Currently, the system keeps track of all tracked points from the start up of each day. Because of this, our algorithm will not react well to buses that change routes mid-day. To mitigate this, we can limit the history of tracked points. For example, we can only store the past 2 full route loops. This way, if a bus changes routes, the route estimation will be updated with a worst case of 2 full route loops.

7.2. Find Buses Passing by

This algorithm finds possible buses that pass by a user's location. The algorithm computes possible buses using these steps:

1. Pick a tracked bus
2. Find all points in known route that corresponds to the tracked bus' estimated route(s).
3. Find closest route point to the user's location.
4. If within acceptable range, add to close buses list.
5. repeat 1-4 for all tracked buses
6. (optional) filter out buses that are far away or filter out buses of same route.

The output of this algorithm is a list of tracked buses that will potentially pass by the user's location. If a tracked bus still has multiple potential bus route names, the front end application will alert the user that an unspecified bus will be passing by and display the possibilities.

Using the tracked bus' most recent coordinates, we can get the estimated arrival times for those buses to the user's location. Without this algorithm, the user would be burdened with searching through all the tracked buses to find those that he is able to ride from his location.

7.3. Find Waypoints

While this algorithm is not included in final version of our project, it has still been implemented and would most likely be included in future versions. The Find Waypoints algorithm is used to find GPS coordinates to feed into Google Maps in order to ensure that the estimated routes follow the routes that the bus will be taking. The following steps are used to create a list of waypoints for a user's location, bus' location pair:

1. Find points for route estimated for the specified bus.
2. Find closest route point to the bus' location.
3. Find closest route point to the user's location.
4. Add points to list between points found in (2) and (3).
5. (optional) Filter out points too clost to each other.

This list will be used in the Google Maps API call that will determine the estimated time of arrival for the specified bus.

8. Hardware

References

[1] "Crawdad: A community resource for archiving wireless data at dartmouth." [Online]. Available: http://crawdad.cs.dartmouth.edu/meta.php?name=rice/ad_hoc_city
[2] "Cta bus tracker." [Online]. Available: http://www.ctabustracker.com/bustime/home.jsp
[3] "Tiramisu, the real-time bus tracker." [Online]. Available: http://www.tiramisutransit.com/t/about
[4] Madelaine Criden, "Recognizing the importance of public transportation for low-income households," 2007.